CS 213 -- Lecture #15

"Late Night Guide to C++"
Chapter 9 pg 240 - 254
MORE ABOUT CLASSES, Part II

Administrative...

- We'll need to start talking about final projects soon!
- Assignment #7 up, due on Thursday

In Our Last Exciting Episode...

- In Part I of our "More Classes" lecture, we ended with the topic of "Declarations within Classes"
- We touched on enumerations
- We touched on static members (variables and functions)
- Today we pick up with classes defined within classes
- These are called "Nested Class Definitions"
- Consider the following:

```
class X {           // X is an arbitrary class
public:
  class Y           // Y is defined in X's public section
  {};
private:            // Z is defined in X's private section
  class Z
  {};
};
```

Nested Class Definitions

- The fact that class `Y` is defined in the public section of class `X` means the following:
  - Objects of class `Y` may be created outside the scope of `X` only if fully qualified name is used (`X::Y`)
- The fact that class `Z` is defined in the private section of class `X` means the following:
  - Objects of class `Z` may only be created within the scope of `X`.
- If Y had been defined within a protected section of class `X`'s definition, objects of type `Y` could be created anywhere in the scope of `X` as well as in the scope of any class derived from `X`.
- Any member variables/functions of Y are accessible according to the normal public, private and protected used in classes.

Nested Class Definitions (cont)

- LNG gives us an example of nested class definitions which revolves around list management.
- Consider the following:

```
class List {    // Abbreviated version of List from LNG
public:
  List():the_head(NULL){}   // standard constructor
  ~List();                  // standard destructor
private:
  class Node {              // Nested definition of Node
  public:
    string the_value;       // the "value" of an element
    Node *next;             // pointer to next element
  };
  Node *the_head;           // the first element
};
```

Demonstration

Nested Class Definitions

Will You Be My Friend?

- Sometimes it is necessary to (or just more convenient) to allow global functions or member functions of another class to have access to private members of a given class.
- Suppose you wanted to keep track of lists of music albums with a Discography class. It might look something like this:

```
class Discography {
public:
  Discography();
  void addAlbum(MyString &anAlbum);
  void setMaxAlbums(int maxAlbums);
  void printAlbums();
  void sortAlbums();
private:
  MyString *albumList;
  int numAlbums, maximumAlbumCount;
};
```

Will You Be My Friend? (cont)

- You may begin to see some familiar concepts:
  - `albumList` is an array of `MyString` instances
  - `setMaxAlbums()` will dynamically allocate space for `albumList`
  - `addAlbum()` will stick another album title at the end of the list
  - `printAlbums()` and `sortAlbums()` work on the private array that gets built during the lifetime of a `Discography` object.
- Two potential problems/annoyances come to mind…
- First, what if I already have a perfectly good piece of reusable code which sorts arrays of `MyString` instances?
- Second, what if there is a very fancy print album function already in existence which searches a database and pulls up pictures of the covers, etc., etc…
- These are both examples of why you might want to selectively open up private data to outside functions.

Will You Be My Friend? (cont)

- The technical term for this is a *friend function*.
- From LNG, page 246:
  - A global function may use the names of the private members of a class (for that is what granting access entails) if a declaration of that function preceded by the keyword `friend` is included in the class definition.
- Now, one way around the issue of an existing library is to simply implement `printAlbums()` and `sortAlbums()` in the `Discography` class in terms of the library functions.
- Another way would be to provide an overloaded definition of the library call (assuming you had control over that source)…
- Suppose I have a function named printAlbumDiscography...

```
void printAlbumDiscography(Discography &disco)
{
  for (int i=0; i<disco.numAlbums; i++)
    LookupAndPrintAlbum(disco.albumList[i]);
}
```

Will You Be My Friend? (cont)

- Having a global function take an instance of Discography as an argument and directly access its private members will only work if we modify our Discography class as follows:

```
class Discography {
public:
  Discography();
  void addAlbum(MyString &anAlbum);
  void setMaxAlbums(int maxAlbums);
  void printAlbums();
  void sortAlbums();

friend void printAlbumDiscography(Discography &);

private:
  MyString *albumList;
  int numAlbums, maximumAlbumCount;
};
```

Will You Be My Friend? (cont)

- Suppose now I wanted to create an Album class.
- The Album class would contain additional information about band members and the year the album was released.
- It might look like this:

```
class Album {  // Quick and dirty Album class
public:
  Album();
  void addBandMember(MyString &aMember);
  void setNumMembers(int memberCount);
  void setAlbumTitle(MyString &theTitle);
  void setAlbumYear(int theYear);
private:
  MyString *bandMembers;
  int numMembers;
  MyString albumTitle;
  int albumYear;
};
```

Will You Be My Friend? (cont)

- Now it would follow that I'd want to be able to pass an `Album` reference to `Discography::addAlbum()` as well as just an instance of `MyString`.
- Suppose, though, that we're happy with keeping the data in the `Discography` class as an array of `MyString` instances (and we don't feel the need to convert it to an array of `Album` instances).
- That's fine, but if `Discography::addAlbum(Album &)` is going to work we either need an `Album::getAlbumTitle()` getter or we're going to need another variant of the *friend* concept:

```
// Add album to the discography by passing an Album ref.
Discography::addAlbum(Album &anAlbum)
{
  // We can only do the following if we're a "friend"
  addAlbum(anAlbum.albumTitle);
}
```

Will You Be My Friend? (cont)

• So, in order for our implementation of addAlbum() to work, we'd need to see *at least* the following:

```
class Album {  // Quick and dirty Album class
public:
  Album();
  void addBandMember(MyString &aMember);
  void setNumMembers(int memberCount);
  void setAlbumTitle(MyString &theTitle);
  void setAlbumYear(int theYear);
friend Discography::addAlbum(Album &);  // Name our friend!
private:
  MyString *bandMembers;
  int numMembers;
  MyString albumTitle;
  int albumYear;
};
```

Will You Be My BEST Friend?

• Sometimes you may find that one class needs access to private data in another class.

• This can be done by declaring an entire class as being a friend in a given class definition.

• Using the example in LNG (and going back to linked lists) we might define a `ListIterator` class to help navigate through a `List`.

• Such a class would need access to the private data in `List`, and would be designated as such in the class definition for `List`:

```
class List {
  friend class ListIterator;
public:
  // member functions and nested class definition
private:
  Node *the_head;
};
```

Final Thoughts

• Assignment #7 is posted, due next Thursday